Diseño de algoritmos recursivos

Isabel Pita

Facultad de Informática - UCM

27 de septiembre de 2021

Objetivos

1. Tipos de recursión.
2. Análisis de la complejidad de algoritmos recursivos. Despliegue de recurrencias.

Introducción

Solución recursiva de un problema:

- Se conoce la solución del problema para un conjunto de datos *simple*. (Caso base o directo)
- Se conoce como resolver el problema para el resto de los datos a partir de la solución del problema para casos más sencillos.(Caso recursivo)

Ejemplos:

- Cálculo de la potencia: $x^0 = 1; x^n = x^{n-1} * x;$
- Cálculo del factorial: $0! = 1; n! = (n-1)! * n;$
- Suma de las componentes de un vector.
- Problema de las torres de Hanoi

Tipos de recursión

Recursión simple (o lineal): cada caso recursivo realiza exactamente una llamada recursiva.

```
using lli = long long int;
lli factorial ( int n ){
  if ( n == 0 ) return 1;
  else return = n * factorial(n-1);   // (n > 0)
}
```

1. Condición del caso directo: $n == 0$
2. Cálculo del caso directo: `return` 1
3. Función sucesor: $s(n) = n - 1$
4. Función de combinación: $n * fact(s(n))$

- Se pueden tener varios casos directos o varias descomposiciones para el caso recursivo. Las alternativas deben ser **exhaustivas** y **excluyentes**.
- Ejemplo: búsqueda binaria.

```
bool buscar(std::vector<int> cont& v,int x,
      int ini,int fin)
{
  if (ini == fin) return false;
  else {
    int m = (ini + fin) / 2;
  if (x < v[m]) return buscar(v,x,ini, m);
  else if (x > v[m] return buscar(v,x,m+1,fin);
  else return true;
  }
}
```

Tipos de recursión simple o lineal.

La recursión simple o lineal puede ser:

- Recursión final o de cola (*tail recursion*). La función de combinación se limita a transmitir el resultado de la llamada recursiva. El resultado será siempre el obtenido en uno de los casos base.
  Ejemplo: Cálculo del máximo común divisor por el método de Euclides.
- Recursión no final. La función de combinación modifica el resultado de la llamada.
  Ejemplo: función factorial.

Tipos de recursión

Recursión múltiple: al menos en un caso recursivo, se realizan varias llamadas recursivas.

```
int fib( int n )
{
  if ( n == 0 ) return 0;
  else if ( n == 1 ) return 1;
  else  return fib(n-1) + fib(n-2);
}
```

Ejemplos. Recursion sobre vectores

Escribir las componentes de un vector:

```
void f (std::vector<int> const& v, int i, int n) {
  if (i == n-1) std::cout << v[i];
  else {
    std::cout << v[i] << ' + ';
    f(v, i+1, n);
  }
}
```

- LLamada inicial: f(v, 0, v.size());
- Dado el vector 2 4 6, escribe 2 + 4 + 6

Ejemplos. Recursión sobre vectores

Escribir las componentes de un vector:

```
void f (std::vector<int> const& v, int i, int n) {
  if (i == n-1) std::cout << v[i];
  else {
    f(v, i+1, n);
    std::cout << ' + ' << v[i];
  }
}
```

- LLamada inicial: f(v, 0, v.size());
- Dado el vector 2 4 6, escribe 6 + 4 + 2

Ejemplos. Recursión sobre números

Escribir los dígitos de un número:

```
void f (int n) {
  if (n < 10) std::cout << n;
  else {
    f(n / 10);
    std::cout << ' + ' << n % 10;
  }
}
```

- LLamada inicial: f(x);
- Dado el número x = 1572 escribe 1 +5+7+2

Ejemplos. Recursión sobre números

Escribir los dígitos de un número:

```
void f (int n) {
  if (n < 10) std::cout << n;
  else {
    std::cout << ' + ' << n % 10;
    f(n / 10);
  }
}
```

- LLamada inicial: f(x);
- Dado el número x = 1572 escribe 2 + 7 + 5 + 1

Análisis de la complejidad de algoritmos recursivos

- Para analizar la complejidad de un algoritmo recursivo debemos analizar la complejidad de lo que se hace en una llamada recursiva, y estimar el número de llamadas recursivas que se realizarán.
- Introducimos el concepto de *ecuaciones de recurrencia*
- Cálculo del factorial.

```
int factorial ( int n ){
  if (n == 0) return 1;
  else return n * factorial(n-1);   // (n > 0)
}
```

- Ecuaciones de recurrencia:

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ c_1 + T(n-1) & \text{si } n > 0 \end{cases}$$

- Multiplicación por el método del campesino egipcio.

```
int prod ( int a, int b )
{
  if ( b == 0 ) return 0;
  else if ( b == 1 ) return a;
  else if (b %2 == 0) return prod(2*a, b/2);
  else if (b %2 == 1) return prod(2*a, b/2) + a;
}
```

- Ecuaciones de recurrencia: ($n = b$ tamaño del problema)

$$T(n) = \begin{cases} c_0 & \text{si } n = 0, 1 \\ c_1 + T(n/2) & \text{si } n > 1 \end{cases}$$

- Números de fibonacci.

```
int fib( int n )
{
  if  ( n == 0 ) return 0;
  else if ( n == 1 ) return 1;
  else return fib(n-1) + fib(n-2);
}
```

- Ecuaciones de recurrencia:

$$T(n) = \begin{cases} c_0 & \text{si } n = 0, 1 \\ T(n-1) + T(n-2) + c_1, & \text{si } n > 1 \end{cases}$$

Despliegue de recurrencias

Método para obtener una *fórmula explícita* del orden de complejidad de una recurrencia.

1. **Despliegue**. Sustituimos las apariciones de T en la recurrencia tantas veces como sea necesario hasta encontrar una fórmula que dependa del número de llamadas recursivas k.
2. **Postulado**. Obtenemos el valor de k que nos permite alcanzar un caso directo y, en la fórmula paramétrica, sustituimos k por ese valor y la referencia recursiva T por la complejidad del caso directo. La fórmula obtenida es la expresión explícita del orden de complejidad.
3. **Demostración**. La fórmula explícita así obtenida sólo es correcta si la recurrencia para el caso recursivo también es válida para el caso directo. Se comprueba demostrando por inducción que la fórmula obtenida cumple las ecuaciones de recurrencia.

- Factorial
  - Ecuaciones

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ c_1 + T(n-1) & \text{si } n > 0 \end{cases}$$

  - Despliegue

$$\begin{aligned} T(n) &= c_1 + T(n-1) \\ &= c_1 + c_1 + T(n-2) \\ &= c_1 + c_1 + c_1 + T(n-3) \\ &\dots \\ &= c_1 * k + T(n-k) \end{aligned}$$

  - Postulado
    El caso directo se tiene para $n = 0$

$$n - k = 0 \Leftrightarrow k = n$$

$$T(n) = c_1 n + T(n-n) = c_1 n + T(0) = c_1 n + c_0$$

    Por lo tanto $T(n) \in O(n)$

Disminución del tamaño del problema por sustracción.

- Cuando:
  1. la descomposición recursiva se obtiene restando una cierta cantidad constante
  2. el caso directo tiene coste constante
  3. la preparación de las llamadas y la combinación de los resultados tiene coste polinómico, entonces la ecuación de recurrencias será de la forma:

$$T(n) = \begin{cases} c_0 & \text{si } 0 \leq n < n_0 \\ a * T(n-b) + c * n^k & \text{si } n \geq n_0 \end{cases}$$

- Se puede demostrar (libro Diseño de programas. Formalismo y Abstracción. R. Peña. Pag: 17):

$$T(n) \in \begin{cases} \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n/b}) & \text{si } a > 1 \end{cases}$$

- Vemos que, cuando el tamaño del problema disminuye por sustracción,
  - En recursión simple ($a = 1$) el coste es polinómico y viene dado por el producto del coste de cada llamada ($c * n^k$) y el coste lineal de la recursión (n).
  - En recursión múltiple ($a > 1$), por muy grande que sea b, el coste siempre es exponencial.

Disminución del tamaño del problema por división.

- Cuando:
  1. la descomposición recursiva se obtiene dividiendo por una cierta cantidad constante
  2. el caso directo tiene coste constante
  3. la preparación de las llamadas y la combinación de los resultados tiene coste polinómico, entonces la ecuación de recurrencias será de la forma:

$$T(n) = \begin{cases} cn^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

- Se puede demostrar (libro Diseño de programas. Formalismo y Abstracción. R. Peña. Pag: 20):

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k * \log n) & \text{si } a = b^k \\ \Theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

- Si $a \leq b^k$ la complejidad depende de n^k que es el término que proviene de $c * n^k$ en la ecuación de recurrencias, y, por lo tanto, la complejidad de un algoritmo de este tipo se puede mejorar disminuyendo la complejidad de la preparación de las llamadas y la combinación de los resultados.
- Si $a > b^k$ las mejoras en la eficiencia se pueden conseguir
  - disminuyendo el número de llamadas recursivas a o aumentando el factor de disminución del tamaño de los datos b, o bien
  - optimizando la preparación de las llamadas y combinación de los resultados, pues, si esto hace disminuir k suficientemente, podemos pasar a uno de los otros casos: $a = b^k$ o incluso $a < b^k$.